


# Creating teaching materials for C++ and CUDA with xeus-cpp

Hristiyan Shterev

# About me

I am Hristiyan Shterev and I`m a high-school student with a strong interest in C++ and systems programming.

I have around four years of experience working with C++, during which I have built a solid foundation in the language.

Some previous projects I`ve worked on include a simple C++ game with the SFML library and a chat system in C# that sends and receives messages within a LAN connection.

Through my internship at Compiler Research, I aim to expand my understanding of interactive execution environments and Jupyter kernels by working with the xeus-cpp project.

Personal Email: hristiyansterev@gmail.com

School Email: hristiyanshterev_20a@schoolmath.eu

GitHub: HrisShterev

Location: Plovdiv, Bulgaria

# Why I chose this project

I chose this project because it makes learning programming more interactive and easier for students.

Using Jupyter Notebook, teaching materials can combine explanations, code examples, and outputs in one place. This allows students to run and modify code instantly, which improves understanding.

Notebooks also reduce the complexity of setting up development environments, letting learners focus more on programming and GPU concepts.


# Fundamental features of xeus-cpp:

- *Interactive C++ Execution* – unlike traditional C++ development where you write, compile, and run whole programs, xeus-cpp allows you to run small pieces of code immediately in a notebook cell. This instant feedback is great for experimenting, debugging, and learning step by step.
- *Notebook Integration* – you can mix explanations, code, and outputs in one document. This is perfect for teaching because students can read theory, see examples, and test them in the same place without switching between different programs.
- *Persistent Environment* – variables, functions, and objects stay in memory as you run more cells, so you can build programs incrementally. You don't need to restart the whole program every time you make a change, which speeds up experimentation and learning.
- Rich Output and Visualization – you can display not just text, but also tables, charts, images, and plots directly in the notebook. This makes it easier to understand complex results, especially for C++ computations or CUDA experiments.

# First steps of the project

**Installing and configuring the development environment**

- Setting up xeus-cpp in Jupyter Notebook
  Installing and building CUDA and verifying the GPU drivers
- Configuring the compiler and required dependencies for C++ and CUDA

**Testing xeus-cpp with C++ and CUDA examples**

- Running simple C++ programs to verify that the kernel executes correctly
- Running basic CUDA examples to confirm GPU compilation and execution
- Making sure memory allocation, kernel execution, and output are working properly

Here is a small example for allocating VRAM using CUDA

```
float* d_ptr = nullptr; // Declare an empty pointer

cudaMalloc((void**)&d_ptr, bytes); // Allocate VRAM

cudaFree(d_ptr); // Free the memory after use
```

# Porting the courses - links

The next step in my project is porting tutorials and courses to xeus-cpp that create that learning environment for students that want to learn low-level programming languages and parallel processing . Here are some links to courses I`m going to use:

- C/C++: Tutorial: Learning resources C/C++
- OpenMP Tutorial: An Introduction to Parallel Programming with OpenMP
- CUDA Tutorial: CUDA by example

# Princeton Research Computing C++ Resources

The C/C++ course is going to include

Modern C++11 features - move semantics, lambdas, smart pointers

- Modern C++ for Computational Scientists - ARCHER 4-part video series

Performance & HPC-portable code - memory layout, hardware abstraction

- Kokkos Online Class (2020) - multi week course

Language reference & modern features - C++17/20, templates, concepts

- The C++ Reference - cppreference.com

C/C++ transition patterns

- Modern C++ for C Programmers - written resource with GitHub exercises

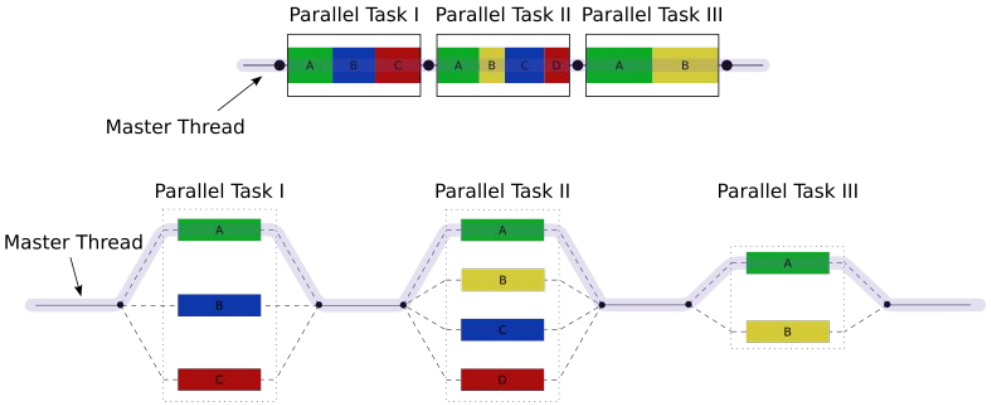
# An introduction to OpenMP

OpenMP is a programming interface that allows you to run parts of your code in parallel across multiple CPU cores. It is designed for shared-memory systems, where all threads can access the same data.

With OpenMP, you don't need to manually create and manage threads. Instead, you use compiler directives, such as #pragma omp, to tell the program which loops or sections of code can run simultaneously. The compiler and runtime then handle splitting the work and managing the threads.

The parallel directive tells OpenMP to run the code inside the following block in parallel. When the program reaches this parallel region, OpenMP creates a team of threads. Any code inside the block can use these threads. When the block ends, the threads are destroyed.

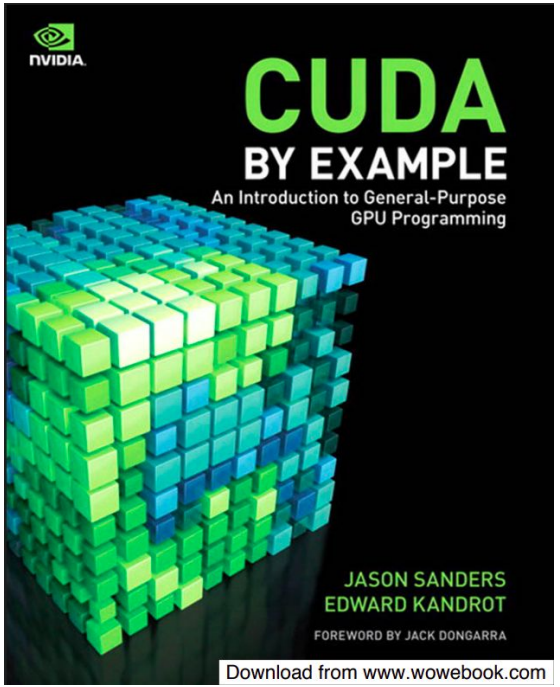
This is an example of fork-join parallelism: threads are "forked" at the start of the parallel region and "joined" at the end.



# “Cuda  by example” course

This course demonstrates simple yet important features for GPU programming. Some examples i`m going to port using Jupyter notebooks are:

- Chapter 3 - Introduction to CUDA C
  - Simple introduction to CUDA
  - Explaining new syntax
- Chapter 4 - Parallel programming in CUDA C
  - introduction to parallel programming + graphs
- Chapter 5 - Thread Cooperation
  - Demonstrating blocks and threads



# Contributing to Xeus-cpp

If the porting work is completed ahead of schedule, the remaining time will be used to make small contributions to xeus-cpp. This could include bug fixes, documentation improvements, or small feature patches.

This is a natural extension of the porting work - having spent time writing and testing C++ notebooks, any rough edges or missing features encountered during porting become direct candidates for patches. This means contributions would be grounded in real usage rather than speculative improvements.

# Thank you!